let and lambda

(**let (** [var1 expr1]
[var2 expr2]
…
[varn exprn] **)**
...body...)

where body refers to args var1, …, varn

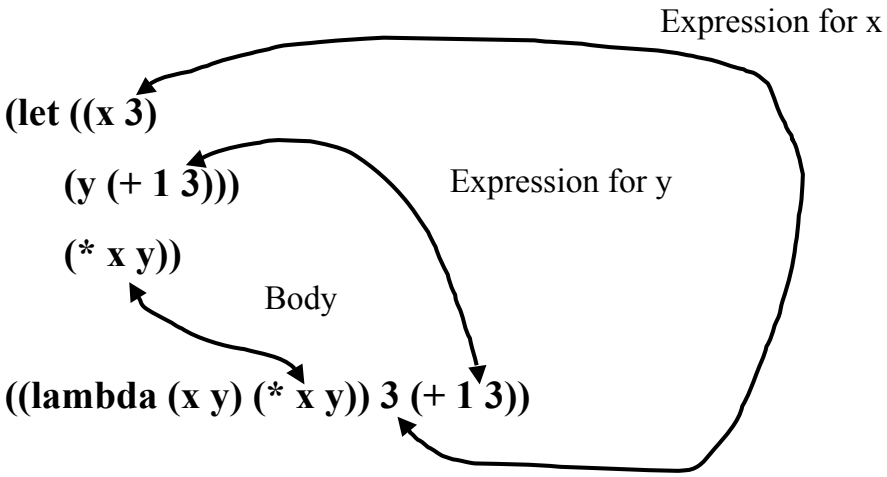
is equivalent to

((**lambda** (var_1 … var_n) … body…)
expr1 … exprn)

so: (**let** ([x (* 4 5)])(+ x x)) as **lambda** is ?

((**lambda** (x) (+ x x)) (* 4 5))

Equivalent Forms

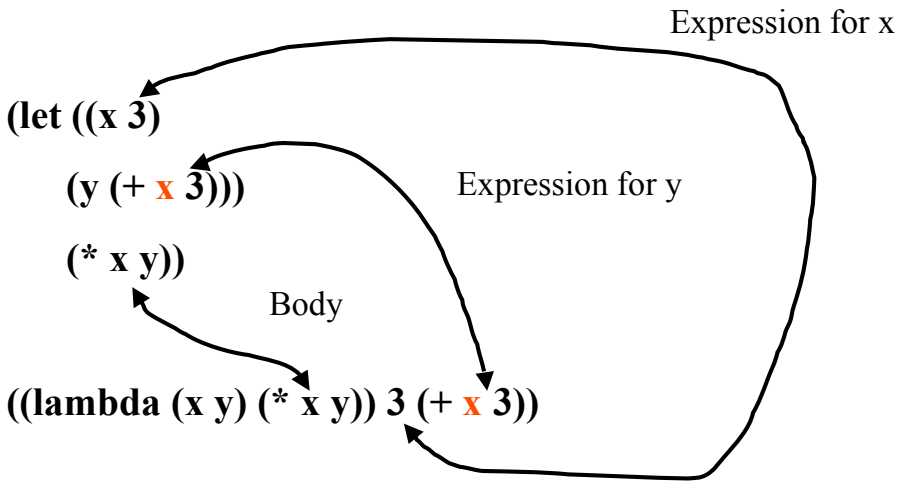


(let ((x 3) (y (+ x 3))) (* x y)) =>
expand: unbound identifier in module
in: x

same for corresponding lambda….

let does not evaluate var-val pairs sequentially!!!!!

Equivalent Forms



recursive vs. iterative control behavior

```
(define fact(lambda (n)
 (if (zero? n) 1 (* n (fact (- n 1))))))
```

```
 (fact 4)
 = (* 4 (fact 3))
 = (* 4 (* 3 (fact 2)))
 = (* 4 (* 3(* 2 (fact 1))))
 = (* 4 (* 3(* 2(* 1 (fact 0)))))
 = (* 4 (* 3(* 2(* 1 1))))
 = (* 4 (* 3(* 2 1)))
 = (* 4 (* 3 2))
 = (* 4 6)
 = 24
```

each call of fact is made with promise to multiply result with value of n at time of call ==>

larger and larger control contexts

```
(define fact-iter (lambda (n)
   (fact-iter-acc n 1)))

(define fact-iter-acc (lambda (n a)
    (if (zero? n) a
     (fact-iter-acc (- n 1) (* n a)]
```

```
   (fact-iter 4)
 = (fact-iter-acc 4 1)
 = (fact-iter-acc 3 4)
 = (fact-iter-acc 2 12)
 = (fact-iter-acc 1 24)
 = (fact-iter-acc 0 24)
 = 24
```

no control context!!

evaluation of operands, not calling of procedures, makes control context grow!!!

procedures as arguments

generalizing various summations as in $\sum_{n=a}^{b} f(n) = f(a) + \ldots + f(b)$

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))
```

```
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))
```

abstract the general pattern:

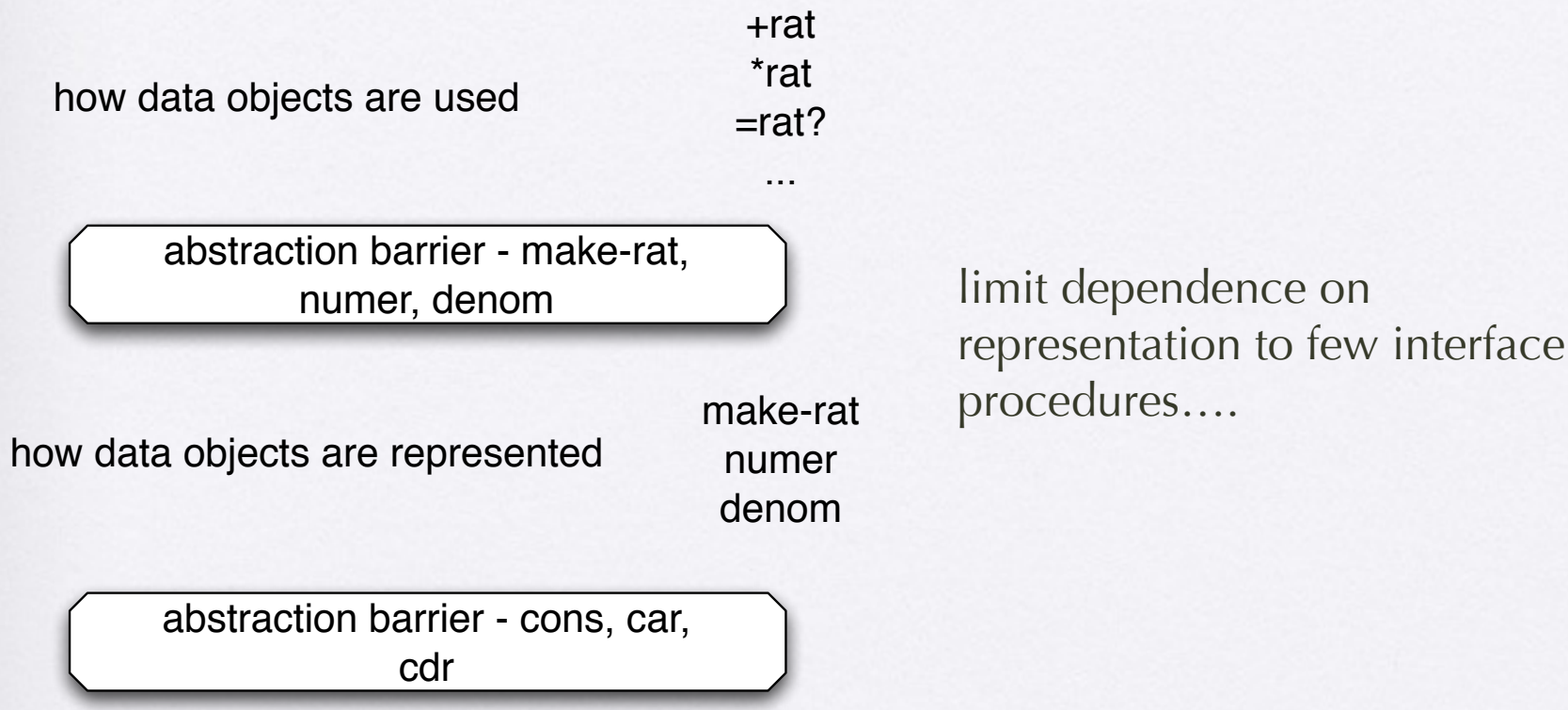
```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

```
(define (inc n) (+ n 1))
(define (identity x) x)
```

```
(define (sum-integers a b)
  (sum identity a inc b))
```

```
(define (sum-cubes a b)
  (sum cube a inc b))
```

data abstraction



limit dependence on representation to few interface procedures….

how are pairs implemented? what **is** a pair?

data abstraction

how are pairs implemented? what **is** a pair?

axiom:
if z is (cons x y), for any x, y, then (car z) is x and (cdr z) is y.

any 3 procedures satisfying this axiom can be used to implement pairs!!!

we could implement cons, car, cdr without any data structures at all --- nothing left but procedures!!!!


```
(define (cons x y)
  (define (dispatch m)
    (cond ((= m 0) x) ((= m 1) y) (else (error "arg not 0 or 1 in cons" m))))
  dispatch)

(define (car z) (z 0))

(define (cdr z) (z 1))
```

more consing and appending

reverse top level of a list: (reverse '{a (b c) d)) => (d (b c) a)

```
(define (reverse lis)   ;;recursive, same effect as built in reverse
  (if (null? lis) '()
      (append (reverse (cdr lis))
              (list(car lis)))))
```

iterative reverse

```
(define (reverse lis)          ;;iterative reverse
  (define (reverse-iter lis x)
    (if (null? lis)
        x
        (reverse-iter (cdr lis)(cons (car lis) x))))
  (reverse-iter lis '()))
```

reverse (and flatten) **all** levels of a list:
(reverse-all '{a ({b (c)}) d)) => (d c b a)

```
(define (reverse-all lis)
  (cond [(null? lis) '()]
        [(not (pair? (car lis)))
                   (append(reverse-all (cdr lis))(list (car lis)))]
        [else (append (reverse-all (cdr lis))
                                   (reverse-all(car lis)))]))
```

sequences as conventional interfaces

expressing functions in terms of applications of several *generalized* functions like *map*, *accumulate*, *filter*, *enumerate*...

```
(define (enumerate-interval low high)
  (if (> low high) '()
      (cons low (enumerate-interval (+ low 1) high))))

(define (enumerate-tree tree)        ;;fringe!!!!
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree)) (enumerate-tree (cdr tree))))))
```

```
(define (sum-odd-squares tree)
(accumulate +
            0
            (map sqr
                 (filter odd?
                         (enumerate-tree tree)))))

(define (list-odd-fib 0 n)
(accumulate cons
            '()
            (filter odd?
                 (map fib
                         (enumerate-interval 0 n)))))
```

objects - set! & local variables

```
(define (make-withdraw balance) ;;balance is local!!
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

```
(define W1 (make-withdraw 100))
 (define W2 (make-withdraw 100))
 (W1 50)=> 50
 (W2 70)=> 30
 (W2 40)=> Insuff..
 (W1 40)=> 10
```

```
(define (make-account balance) ;;balance is local!!
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request" m))))
  dispatch)
```

```
(define acc (make-account 100))
((acc 'withdraw) 50) => 50
((acc 'withdraw) 60) => Insuff.
((acc 'deposit) 40)  => 90
((acc 'withdraw) 60) => 30
```

examples of local state and independent objects

```
(define count1
   (let ([count 0])
     (lambda ()
     (set! count (+ 1 count))
     count)))
```

```
(count1) => 1
(count1) => 2
(count1) => 3 ...
```

starting to count at 0:


```
(define count
   (let([next 0])
     (lambda ()
       (let ([v next])
         (set! next (+ next 1))
         v))))
```

```
(count) => 0
(count) => 1
(count) => 2 ...
```

```
(define make-counter
  (lambda ()
    (let([next 0])
      (lambda ()
        (let([v next])
          (set! next (+ 1 next))
          v)))))
```

```
(define c1 (make-counter))
(define c2 (make-counter))
(c1) => 0
(c2) => 0
(c2) => 1
(c2) => 2
(c1) => 1
```

stack factory

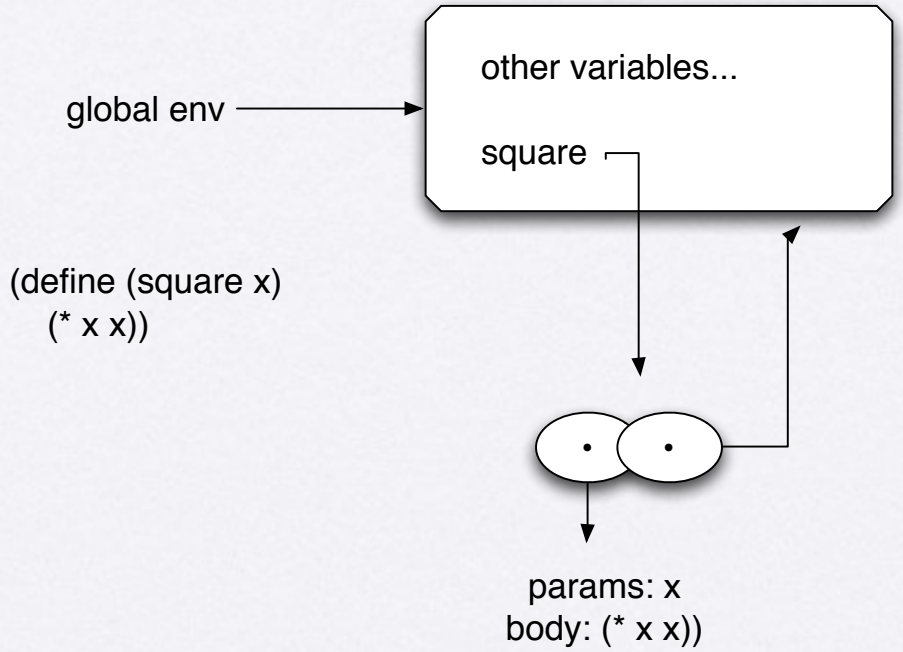
```
(define (make-stack)
   (let([st '()])
     (lambda (msg . args) ;;because push! has arg...
       (cond
         [(eqv? msg 'empty?) (null? st)]  ;;eq? also ok
         [(eqv? msg 'push!)
          (set! st (cons (car args) st))]
         [(eqv? msg 'pop!) (set! st (cdr st))]
         [(eqv? msg 'top) (car st)]
         [else "oops!"]))))
```

```
(define stack1 (make-stack))
(define stack2 (make-stack))
(stack1 'empty?) => #t
(stack1 'push! 'hi)
(stack1 'empty?) => #f
(stack1 'push! 'ouch)
(stack1 'top) => 'ouch
(stack2 'push! (stack1 'top))
(stack2 'top) => 'ouch
```

we could now replace the list implementation by a vector without affecting the API...

environment model of evaluation

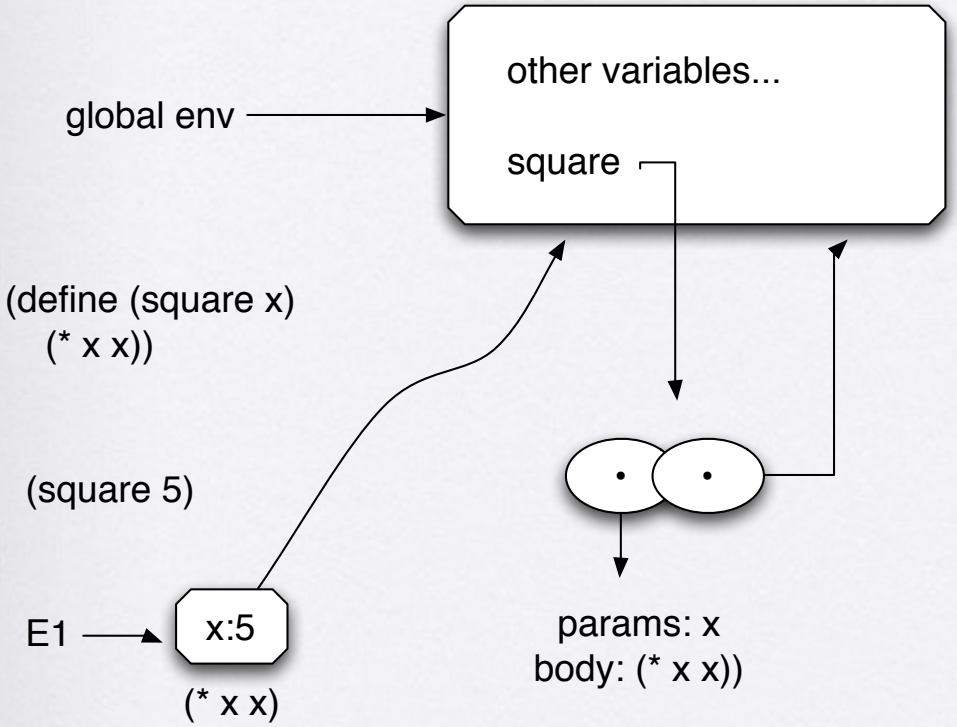
a procedure results from evaluating a lambda expr: a procedure is a pair of code and a pointer to the env in which the lambda expr was evaluated



rule 1: a lambda expression is evaluated relative to a given env: a new procedure object is made, combining the text/code of the lambda expr with a pointer to the env of evaluation….

environment model of evaluation

rule 2: a procedure object is applied to arguments by making a frame, binding formal parameters to actual arguments, and then evaluating the procedure body in the context of the new env made. The new frame has as enclosing env the env part of the procedure object being applied...



rule 2.5: to evaluate (set! var val) in an env, find the 1. frame in env that contains a binding of var and change that frame to bind var to val...

delaying evaluation

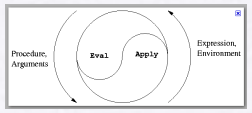
how to do this?
suppose we want to define (*freeze* expr) as (lambda () expr) so that expr is not evaluated
but when we say (define promise (freeze (+ 10 20)) we still get 30 .. :-(

we need a special form, not a procedure!!! (require mzlib/defmacro) or:

```
(define-syntax freeze
  (syntax-rules ()
    [(_ a ...)
     (lambda () a ...)]))
```

```
(define (thaw thunk)
    (thunk))
```

now when we evaluate (freeze (+ 10 20)) we get a procedure of no args, say *promise*, and when we (thaw promise) we get 30..

eval  apply

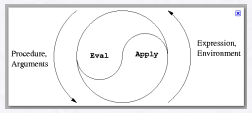
• the evaluator should be as independent as possible from the actual **syntax** of the interpreted language ---- use data abstraction

• (eval expr env) ==>

eval calls itself and apply….

• if expr is self-evaluating (i.e. number or string), eval just returns expr; variables are looked up in env
• procedure application: eval recursively evaluates operator and operands and gives them to apply

• special forms:
definition or assignment: eval computes new value and changes env
if expr: eval consequent if pred is true, else eval alternative
cond expr: changed into nested ifs, then eval
lambda: putting together parameters and body with env of evaluation
etc etc...

eval  apply

- (apply proc args) ==>
  as in (apply + '(1 2 3 4 5)) ...

  - distinguishes between primitive and compound procedures

  - compound procedure:
    sequentially eval exprs in the body;
    env for evaluation results from extending the procedure's base env with a frame binding params to args

  - apply calls eval to deal with compound procedures...

- note: the evaluator does not use **let**, **set!**, **begin**, **internal definitions**, etc. but supports them in the interpreted language!!!!

- this demonstrates that an interpreter supporting certain features can be written in a language without these features!!! bootstrapping!!!

resolution, cont.

- If clause C_1 contains p and C_2 contains ¬p, then the **resolvent of C_1 and C_2 on p** is a clause containing all <u>other</u> elements of C_1 and C_2.
- Principle of (propositional) resolution:

$\therefore$ $((\mathbf{p} \vee \alpha) \wedge (\mathbf{\neg p} \vee \beta)) \rightarrow (\alpha \vee \beta)$

¬p → α, p → β, and since p ∨ ¬p:
α ∨ β, i.e ¬ α → β.
…. transitivity of implication

- algorithm: to derive α from premisses S:

convert S and ¬α to clausal form (and aim for a contradiction)
repeat
    <u>pick</u> p and C_1, C_2 that can be resolved on p
    simplify resolvent by eliminating duplicates
    remove resolvent if it has both a q and ¬q
    add resolvent to original set if not there
if the empty clause results, S implies α.

conversion to clausal form

- e.g. $\{c \rightarrow s, \neg g \rightarrow d, \neg g \vee c\}$ implies $\neg s \rightarrow d$???

$\neg c \vee s, g \vee d, \neg g \vee c$, neg of conclusion: $\neg s \wedge \neg d$

$C_1 = \{\neg c, s\}, C_2 = \{g, d\}, C_3 = \{\neg g, c\}, C_4 = \{\neg s\}, C_5 = \{\neg d\},$

$(((C_2 * C_5) * C_3) * C_1) * C_4) = \square$ = empty clause (successful derivation)

- e.g. $\{\neg p \rightarrow r, p \rightarrow s, r \rightarrow q, s \rightarrow \neg t, t\}$ implies q??

$C_1=\{p,r\}, C_2=\{\neg p,s\}, C_3=\{\neg r,q\}, C_4=\{\neg s,\neg t\}, C_5=\{t\}, C_6=\{\neg q\}.$

$\ldots (((C_3 * C_6) * C_1) * C_2) * C_4) * C_5 =$ empty clause $\square$

- e.g. prove $p \vee \neg p$: negate: $\neg p \wedge p$; $\{\neg p\}, \{p\}$ resolves to empty clause.

Conversion to clausal form is a bit trickier in fol:

$$\neg \forall \upsilon \, \Phi \Leftrightarrow \exists \upsilon \, \neg \Phi, \quad \neg \exists \upsilon \, \Phi \Leftrightarrow \forall \upsilon \, \neg \Phi$$

standardize variables apart: $\forall x Fx \wedge \exists x Gx$ becomes $\forall x Fx \wedge \exists y Gy$

more lists

define last(Item,List) so that
last(joe,[mary,jim,joe]) is true
last(X, [mary,jim,joe]) => X = joe

with append:

```
last(Item,List) :-
  append(_,[Item],List).      %% ;-)
```

without append:

```
last(Item,[Item]).
last(Item,[First|Rest]) :-
  last(Item,Rest).
```

define reverse(X,Y) :

```
reverse([],[]).
reverse([First|Rest],Reversed) :-
  reverse(Rest, ReversedRest),
  append(ReversedRest,[First], Reversed).

palindrome(List) :- reverse(List,List).
```

flatten a list

```
flatten(List, FlatList) so that

?- flatten([a,b,[c,d],[[e,f],g],h,[],[[[j]]],k],L) =>
L = [a,b,c,d,e,f,g,h,j,k]



flatten([H | T], FlatList) :-
     flatten(H, FlatHead),
     flatten(T, FlatTail),
     append(FlatHead,FlatTail,FlatList).


flatten([], []).

flatten(X, [X]).  % flatten a non-list
```

evenlength & oddlength

```
%  evenlength([a,b,c,d]).
% evenlength([a,b,c]).



evenlength([]).

evenlength([First | Rest]) :-
   oddlength(Rest).

oddlength([_]).

oddlength([First | Rest]) :-
  evenlength(Rest).
```

controlling backtracking with cut -- !

- coming from left, predicate ! always succeeds
- coming from right,
  ! fails,
  whole rule fails,
  other rules with same predicate fail

suppose function f(X,Y) such that Y = 0 if X ≤ 0, else Y = 1

```
f(X, 0) :- X =< 0, !.
f(X, 1) :- X > 0.
```

without !, if 1. rule succeeds and subsequent backtracking happens, 2. rule will be tried *uselessly*…

===> green cut:

program still works without it but inefficient

controlling backtracking with cut -- !

```
suppose function f(X,Y) such that Y = 0 if X ≤ 0, else Y = 1

f(X, 0) :- X =< 0, !.
f(X, 1). % this is only done when 1. rule has failed
```

without !, if 1. rule succeeds and subsequent backtracking happens, 2. rule gives *wrong* result…

===> red cut:

program gives wrong result without !

```
max(X,Y,X) :- X >= Y, !.     max(X,Y,Y).

mem1(X, [X|T]) :- !.
mem1(X, [_|T]) :- mem1(X,T).
% if X occurs several times, mem1 only answers once…
```

cut and fail to state exceptions

- built-in **fail** fails immediately, i.e. causes backtracking

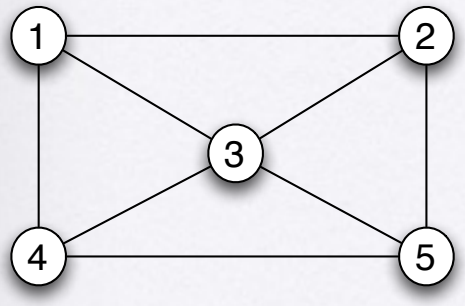
```
?- mortal(X), write(X), fail.  % writes all mortals...
```

- **cut** prevents backtracking

so??? **exceptions to general rules**

```
i like all chips:   like(i,X)  :-  chips(X).

but I don't like Doritos:

like(i,X)  :-  dorito(X),!,fail.
like(i,X)  :-  chips(X).

chips(X)  :-  chip_kind_x(X).
chips(X)  :-  dorito(X).
chips(X)  :-  chip_kind_y(X).

chip_kind_x(a).                    i like a,c,d but not b!!!
dorito(b).
chip_kind_x(c).
chip_kind_y(d).
```

graph




```
edge(1,2).
edge(1,4).
edge(1,3).
edge(2,3).
edge(2,5).
edge(3,4).
edge(3,5).
edge(4,5).
```

Express the fact that the edges are bi-directional without adding more facts like edge(2,1). etc.:

```
connected(X,Y) :- edge(X,Y) ; edge(Y,X).
```

```
?- path(1,5,P).
P = [1,2,5] ;
P = [1,2,3,5] ;
P = [1,2,3,4,5] ;
P = [1,4,5] ;
P = [1,4,3,5] ;
P = [1,4,3,2,5] ;
P = [1,3,5] ;
P = [1,3,4,5] ;
P = [1,3,2,5] ;
no
```

```
path(A,B,Path) :-
       travel(A,B,[A],Q),
       reverse(Q,Path).

travel(A,B,P,[B|P]) :-
       connected(A,B).
travel(A,B,Visited,Path) :-
       connected(A,C),
       C \== B,
       \+member(C,Visited),
       travel(C,B,[C|Visited],Path).
```